SCR.1.2 TD 05 ⊥ :

Le shell

Substitutions - Expressions conditionnelles

Commandes composées

Pour chacun des mécanismes suivants, on en lit la description, on applique sur les exemples fournis et on complète tous les pointillés.

L'affectation d'une variable peut se faire par une commande simple de la forme :

```
name=value
```

Avec :

```
name is a word consisting only of alphanumeric characters and underscores,
and beginning with an alphabetic character or an underscore.  Also referred
to as an identifier.
```

I. Substitutions.

1. Si `name` est le nom d'une variable, alors `$name` sera substitué par `value`

   `x=pop`
   Alors `echo x` ; affiche : ....., et `echo $x` ; affiche : .....

   `x=5+7` ; Alors `echo $x` ; affiche : .....

   `PWD` est une variable du shell qui retient le chemin complet vers le répertoire de travail courant :

   `PWD: The current working directory as set by the cd command.`

   On veut afficher la chaîne de caractères :

   `Working directory: /export/home/students/quidam/SCR/`

   où ce qui vient après les `:` doit se substituer, à chaque fois, par le chemin complet du répertoire courant.

   Une ligne de commande qui le fait : .....

2. `Command substitution allows the output of a command to replace the command name.  Form: $(command)`

   La commande `file` permet de renseigner sur le type du contenu d'un fichier.

   ```
   $ file td6.ASR.1.1.tex td6.ASR.1.1.dvi DIRECT/
   td6.ASR.1.1.tex: LaTeX 2e document, ASCII text
   td6.ASR.1.1.dvi: TeX DVI file (TeX output 2015.11.03:1803\213)
   DIRECT/:         directory
   ```

   On me remet un ficher `list` contenant une liste de noms d'éléments dont je dois vérifier le type du contenu. Je veux avoir en un seul affichage le type du contenu de chacun des éléments dont le nom est enregistré dans le fichier `list` :

   Une ligne de commande qui le fait : .....

3. `Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result.`
   `The format for arithmetic expansion is: $((expression))`

   ```
   Shell variables are allowed as operands; parameter  expansion  is  performed
   before the expression is evaluated. Within an expression, shell variables may
   also be referenced by name without using the parameter expansion syntax.
   A shell variable that is null or unset evaluates to 0 when referenced by name
   ```

```
without using the parameter expansion syntax.
The value of a variable is evaluated as an arithmetic expression when it is
referenced, or when a variable which has been given the integer attribute
using declare -i is assigned a value. A null value evaluates to 0.  A shell
variable need not have its integer attribute turned on to be used in an expression.
```

`x=5+7` (cf. 1) Alors `echo $x` ; affiche : .....

`x=$((5+7))` Alors `echo $x` ; affiche : .....

`y=5 ; z=-7`

`x=y*z` ; alors `echo $x` ; affiche : .....

`x=$((y*z))` ; alors `echo $x` ; affiche : .....

II. Expressions conditionnelles.

Les expressions conditionnelles sont utilisées par la commande :

```
[[ cond_expr ]] returns a status of 0 or 1 depending on the evaluation (True or
False) of the conditional expression cond_expr. Arithmetic expansion is performed
on the words between the [[ and  ]].
```

`cond_expr` est composée à partir d'expressions primaires qui seront étudiées en TP, et dont voici des exemples :

`-a file` : s'évalue à True si `file` est un élément du système de fichiers.

`-d file` : s'évalue à True si `file` est un répertoire.

`-f file` : s'évalue à True si `file` est un fichier régulier.

`string1 == string2` : s'évalue à True si les deux chaînes de caractères `string1` et `string2` sont égales.

`string1 != string2` : You got it !

`string1 < string2` : s'évalue à True si la chaîne de caractères string1 se trie avant la chaîne de caractères string2, selon l'ordre lexicographique (l'ordre du dictionnaire).

`string1 > string2` : You got it !

```
arg1 OP arg2
OP is one of -eq, -ne, -lt, -le, -gt, or -ge. These arithmetic binary operators
return true if arg1 is equal to, not equal to, less than, less than or equal to,
greater than, or greater than or equal to arg2, respectively.
Arg1 and arg2 may be positive or negative integers.
```

Put all together :

1. `[[ myrmidon < myriade ]] ; echo $? ;` affiche : .....
2. `[[ 4567 < 5 ]] ; echo $? ;` affiche : .....
3. `[[ patate == potato ]] ; echo $? ;` affiche : .....
4. `[[ patate -eq potato ]] ; echo $? ;` affiche : ..... et `patate=9 ; potato=7 ; [[ patate -lt potato ]] ; echo $? ;` affiche : .....

III. Commandes composées.

1. `if list; then list; [ elif list; then list; ] ... [ else list; ] fi`

   ```
   The if list is executed. If its exit status is zero, the then list is executed.
   Otherwise, each elif list is executed in turn, and if its exit status is zero,
   the corresponding then list is executed and the command completes. Otherwise,
   ```

```
the else list is executed, if present. The exit status is the exit status of
the lastcommand executed, or zero if no condition tested true.
```

Pour le moment, on restreint les `if list` à être des commandes de la forme `[[ cond_expr ]]` (*cf.* **II.**)

(a)
```
if [[ myrmidon < myriade ]]
> then echo "myrmidon sorts before myriade"
> else echo "myrmidon does not sort before myriade"
> fi
```
affiche : .....

(b) On lance deux fois de suite :
```
if [[ -d VROOM ]] ; then cd VROOM ; else mkdir VROOM ; fi
```
What will happen ?

2.
```
for name in word(s) ;  do list ; done
The list of word(s) following in is expanded, generating a list of items.
The variable name is set to each element of this list in turn, and the do list
is executed each time.
```

(a) `for i in "1 2 3" ; do echo $i ; done ;` s'exécute ... fois et affiche :
???

(b) `for i in 1 2 3 ; do echo $i ; done ;` s'exécute ... fois et affiche :
???

(c) `for i in seq 3 ; do echo $i ; done ;` s'exécute ... fois et affiche :
???

(d) `for i in $(seq 3) ; do echo $i ; done ;` s'exécute ... fois et affiche :
???

(e) `for i in $(seq 3) "1 2 3" ; do echo $i ; done ;` s'exécute ... fois et affiche :
???

```
seq [OPTION]... FIRST INCREMENT LAST
Print numbers from FIRST to LAST, in steps of INCREMENT.
If FIRST or INCREMENT is omitted, it defaults to 1.
```

3.
```
while list-1; do list-2; done
The while command continuously executes the list list-2 as long as the last
command in the list list-1 returns an exit status of zero.
```

Pour le moment, on restreint les `list-1` à être des commandes de la forme `[[ cond_expr ]]` (*cf.* **II.**)

`z=0 ; while [[ z -le 3 ]] ; do echo $z ; z=$((z+1)) ; done ;` s'exécute ... fois et affiche :
???

4. `for (( expr1 ; expr2 ; expr3 )) ; do list ; done`
   `First, the arithmetic expression expr1 is evaluated. The arithmetic expression expr2 is then evaluated repeatedly until it evaluates to zero. Each time expr2 evaluates to a non-zero value, list is executed and the arithmetic expression expr3 is evaluated. If any expression is omitted, it behaves as if it evaluates to 1.`

   (a) `for (( t=5 ; t<=14 ; t=t+2 )) ; do echo $t ; done ;` s'exécute ... fois et affiche :
   ???

   (b)
   ```
   for (( i=1 ; i<4 ; i++ ))
   > do for (( j=1 ; j<4 ; j++ ))
   > do echo -n $((i*j)) ; echo -n " "
   > done
   > echo -e "\n"
   > done
   ```
   La commande `echo -n " "` sera exécutée ... fois.
   La commande `echo -e "\n"` sera exécutée ... fois.
   La commande composée affiche :
   ???